



Testing web applications

By Randal L. Schwartz
merlyn@stonehenge.com
www.stonehenge.com/merlyn/
version 1.8 of 17 Dec 2007

This document is copyright 2007 by Randal L. Schwartz, Stonehenge Consulting Services, Inc.
This work is licensed under Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License
<http://creativecommons.org/licenses/by-nc-sa/3.0/>



What we're talking about

- Why test?
- Perl's testing environment
- Perl's most popular browser emulator
- Mixing the two



Why test?

- Test while coding: test-driven development
- Test for acceptance during integration: make sure other components work to spec
- Test before deploying: don't break the live site!
- Test during maintenance: add bug reports to the test bed



Perl's Testing Environment

- The “test harness” concept
- Test::More
- The prove tool



The “test harness” concept

- Test harness runs one or more test programs
- Test programs are expected to generate a series of “OK” or “Not OK” messages
- Messages are numbered, so we can see if any got dropped or mis-sequenced:
1..3
ok 1
not ok 2
ok 3



Test harness summarization

- Uses the first test program output to determine the number of remaining tests (1..3)
- Helpful if the test program aborts early
- Does the math: (33% failure)
- Over time, the protocol was updated
 - 1..3 can appear last instead of first
 - comments are ignored (# to end of line)
 - test names (ok 1 - reading file)
- Popular TAP protocol now has C libs



In the old days

- Handwritten “ok”/”not ok” tests:
print “not ” if $1 + 2 \neq 3$;
print “ok 7\n”;
- What a pain
- Renumbering the tests to insert one was a mess
- Testing should be easy, or people won’t do it!



Somewhere recently

- Test.pm
- `ok(1 + 2 == 3);`
- printed “ok N” or “not ok N”, increasing N
- No more test counting!
- Also, wasn't standard
 - Had to be CPAN installed
 - But generally needed to install other things
- Then along came the testing cabal



Test::More

- Upward compatible with Test.pm
- Permits labels and good diagnostics
- `is(1 + 2, 3, '1 + 2 = 3');`
- Many modules now migrating to Test::More
- Test::More is standard with 5.8, but works all the way back to 5.5 (installable from CPAN)



Basics of Test::More

- Bring it in:
use Test::More tests => 7; # 7 tests
use Test::More no_plan; # during development
- Use the uncounted version during debugging
- Be sure to count and update before a release



Basic tests

- `ok($boolean, 'test name');`
good if true
- `is($computed, $reference, 'test name');`
good if match
- `isnt($computed, $reference, 'test name');`
- When possible, use `is/isnt` instead of `ok`
- Better diagnostics when something goes wrong



Regular expression tests

- `like($computed, qr/reference/, 'test name');`
- `unlike($computed, qr/reference/, 'test name');`
- Older Perl without `qr//` may require more futzing



Flexible comparisons

- `cmp_ok($this, 'eq', $that, 'test name');`
- 'eq' can be any comparison operator ('ne', '>=' and so on)
- Can be used for “nearly” floating point approximations:
`cmp_ok( abs(sqrt(69)**2 - 69), '<=', 1e-4, 'sqrt 69 squared is nearly 69');`



Can we can this can?

- `can_ok(Class::Name, @method_list);`
- `can_ok($object, @method_list);`
- Useful for verifying good imports:
`can_ok(__PACKAGE__, @import_list);`



Isa Thisa OK-a?

- `isa_ok(Class::Name, @class_list);`
- `isa_ok($object, @class_list);`
- `isa_ok` also supports built-in classes:
`isa_ok($some_reference, 'ARRAY');`



Roll your own

- `pass( 'test name' );`
- `fail( 'test name' );`
- docs say “use these very very very sparingly”
- Better to make the tests read like what they are testing
- Create a local subclass of `Test::Builder`, and use it for your projects



Being verbose

- `diag(@some_messages);`
- Automatically commented at the beginning of each line
- Useful to provide additional verbosity if the tests are being run verbosely
- Writes to a private copy of `STDOUT`, even if your `STDOUT` and `STDERR` have been mangled by the test



Module tests

- `BEGIN { use_ok( 'Some::Module', @imports) }`
- Like “use Some::Module @imports”, but trapping errors
- Do this in BEGIN so imports are early enough
- `require_ok($module);`
- `require_ok($file);`



Conditional tests

- Sometimes, we don't aim for 100%
- Skipped tests for those that aren't run this time
- Todo tests for those tests that are run but expected to fail



Skipped Tests

- SKIP: {
 skip \$why, \$show_many if \$condition;
 test; test; test;
}
- If \$condition is true, says “skipping \$show_many for reason \$why”
- The count keeps the total count consistent, even if the tests are skipped
- Yes, this means you must actually count tests



Todo Tests

- `TODO: {
 local $TODO = $why;
 test; test; test;
}`
- Test success/fail is inverted in a TODO block
- Failed tests are good
- Successful tests are “unexpectedly succeeding”
- The message should give a hint as to why



And there's `todo_skip`

- `TODO:`
 - `todo_skip $why, $show_many if $condition;`
 - `test; test; test;`
 - `}`
- Like a `TODO`, but they really are skipped instead of running to see if they worked.
- Use `TODO` if you haven't finished programming
- Use `SKIP` if the environment might not be right for testing



Structured Comparisons

- `is_deeply($this, $that, 'test name');`
- Deep structure comparison
- Might want to use `Test::Differences` or `Test::Deep` for more complete diagnostics when differences exist



Conditional testing

- Sometimes, a test file won't work for now
- Perform a meta-test, and based on that, skip the entire file:

```
use Test::More;
```

```
if ($<) { # not running as root
```

```
    plan skip_all => 'these tests must run as root';
```

```
} else { # running as root
```

```
    plan tests => 5; # same args as Test::More
```

```
}
```




Sample tests

- From my File::Finder distro, testing protocol:
require_ok('File::Finder');
can_ok('File::Finder',
 qw(new as_wanted as_options in collect));
isa_ok(my \$f = File::Finder->new, 'File::Finder');
isa_ok(\$f->as_wanted, 'CODE');
isa_ok(\$f->as_options, 'HASH');



More sample tests

- Also from File::Finder:
is_deeply(
 [File::Finder
 ->contains('AvErYuNIIkElYsTrInG')
 ->in('.')],
 [“./” . __FILE__],
 'files with a very unlikely string');



Conditional skipping

- Also from File::Finder:
use Test::More;
BEGIN {
 eval { require Test::Distribution };
 plan skip_all =>
 'Test::Distribution not installed' if \$@;
}
\$ENV{TEST_VERBOSE} or
 plan skip_all =>
 'Dist tests selected only in verbose mode';



The prove tool

- In the early days, the Test::Harness was available from a cryptic line in the Makefile for a module
- But how to run things away from a module?
- After all, testing is more than modules
- Enter “prove”
- Contained in the latest Test::More distros



Simple invocations of “prove”

- Run all *.t files in alpha order:
prove
- Enable verbosity:
prove -v
- Add to the @INC path:
prove -v -I../lib
- Run only one test:
prove -v -I../lib 04-bigstuff.t



Other cool options

- Recursively invoke:
prove -r -v -l../lib
- -l is like -l lib
- -t is taint warning mode
- -T is taint fatal mode



Perl's Browser Emulator

- WWW::Mechanize
- Inspired by WWW::Automate
- Built on (inherits from) LWP::UserAgent
- Adds in HTML::Form processing
- Understands “forward”/”backward” like a browser
- Can look for links based on regex on link text or URL



Simple instantiation

- Create a mech object:
use WWW::Mechanize;
my \$mech = WWW::Mechanize->new;
- Takes same args as LWP::UserAgent
- Defaults a few things differently
 - cookie_jar - defaults to a memory jar
 - agent - name is mech-like



Fetching pages

- Fetch the URL:
`$mech->get($url);`
- Was it successful?
`if ($mech->success) { ... }`
- Look at the content:
`$mech->content`
- HTML pages are automatically parsed for links and forms



Following a link

- `$mech->follow_link(@link_spec);`
- `link_spec` contains one or more of
 - `text => 'string'`
 - `text_regex => qr/regex/`
 - `url => 'string', url_regex => qr/regex/`
 - `url_abs => 'string', url_abs_regex => qr/regex/`
 - `tag => 'a', tag_regex => qr/^(a|frame)$/`
 - `n => $count` (which one if many match)



Just finding the links

- Find the link of interest:
 - `my $link = $mech->find_link(@link_spec);`
 - `my @links = $mech->find_all_links(@spec);`
 - `my @links = $mech->links;`
- Returns WWW::Mechanize::Link objects
- Useful methods: `url`, `text`, `tag`, `url_abs`
- Replace `$mech->get($mech->find_link(...))` with `$mech->follow_link(...)`



Finding images

- `$mech->find_image(@link_spec)`
- `$mech->find_all_images(@link_spec)`
- `@link_spec` may include `alt`, `alt_regex`
- Returns `WWW::Mechanize::Image` objects
- Useful methods: `url`, `tag`, `url_abs`, `height`, `width`, `alt`



Form processing

- Return all forms as HTML::Form objects:
`$mech->forms`
- Dump the form (good for debugging):
`$form->dump`
- Select a particular form and return it:
 - `$mech->form_number($n)`
 - `$mech->form_name($name)`



Stuff a form with values

- Works on the current form
- `$mech->field($name, $value)`
- `$mech->select($multiple_select_field, \@values)`
- `$mech->set_fields(...)`
 - `name => $value`
 - `name => [$value1, $value2, $value3, ...]`



Nameless stuffing

- `$mech->set_visible(...);`
 - `$value` (next visible item)
 - `[type => $value, $value2, ...]`
- `type` is one of `text`, `password`, `hidden`, `textarea`, `file`, `image`, `submit`, `radio`, `checkbox`, or `option`
- Helpful if you don't want to care what the fieldnames are



Submitting a form

- `$mech->submit`
- `$mech->submit_form(...)`
 - `form_number => $n`
 - `form_name => $name`
 - `fields => { f1 => v1, f2, => v2 ... }`
 - `button => $button_name`
 - `x => $x, y => $y`



Mixing the two

- Let's "test" a website
- We'll pick `search.cpan.org`
- Start by creating a virtual browser:
use `WWW::Mechanize`;
`my $a = WWW::Mechanize->new;`
- Create the test environment:
use `Test::More` `no_plan`;



Fetch the top-level page

- Fetch, and combine with an OK:
`$a->get("http://search.cpan.org");`
`ok($a->success, "fetched /");`
- This just says "good/bad"
- Better test might be:
`is($a->status, 200, 'fetched /');`
- Then the status will be displayed if it breaks



Back up a step

- But wait! Do we have a mech object?
`isa_ok(my $a = WWW::Mechanize->new,
 'WWW::Mechanize');`
- Now we're sure before we do additional things
- Although this is probably an overkill test
- But then it's a free test that should always work!



Is the title good?

- Let's verify the proper page came back:
like(`$a->title,`
 `qr/The CPAN Search Site/,`
 `'/ title is good'`);
- Of course, this depends on knowing some things about the site
- Important to keep the tests checked in with the site code to stay in sync



What about the FAQ link?

- Verify that the FAQ link is good:
`ok($a->follow_link(text => 'FAQ'),
 'follow FAQ link');`
- This find a link with the text of “FAQ”
- Return code tested by OK verifies that such a link was found



Is it really the FAQ page?

- Verify the page was actually fetched:
`is($a->status, 200, 'fetched FAQ page');`
- See if the content looks sensible:
`like($a->content,
qr/Frequently Asked Questions/,
'FAQ content matches');`
- Now go back to the previous page:
`$a->back;`
- No need to test “back” success



What if the FAQ link fails

- We could get erroneous tests
- `$a->status` and `$a->content` will be against the original fetch
- Things that should fail, will succeed
- Things that should succeed, will fail in a cascading way
- So we need to hook in a SKIP for best testing



Adding the conditional tests

- Revised code:
SKIP: {
 ok(\$a->follow_link ...)
 or skip “missing FAQ link”, 2;
 is(\$a->status ...)
 or skip “bad FAQ fetch”, 1;
 like(\$a->content ...);
 \$a->back;
}
- Now we get sensible test branching



Testing form fillout

- Let's search for author PETDANCE:
`ok($a->submit_form(
 form_number => 1,
 fields => {
 query => 'PETDANCE',
 mode => 'author',
 }), 'look for author PETDANCE');`
- See if it worked:
`is ($a->status, 200, 'found PETDANCE');`



Testing the result

- Verify proper page response:
like(\$a->content, qr/Andy Lester/,
 'found Andy Lester');
- Reset back, since it worked:
\$a->back



But that's a skip block

- Just like before:
SKIP: {
 ok(\$a->submit_form ...) or skip "...", 2;
 is(\$a->status, 200, ...) or skip "...", 1;
 like(\$a->content, ...);
 \$a->back;
}
- That should be a common pattern now



What do we now know?

- The site is up
- The home page looks reasonable
- The FAQ link goes to the right place
- The searching system is working
- The searching system returns the right result
- All with a dozen lines of code
- With enough of these tests, we can sleep at night, knowing all is well



Test::WWW::Mechanize

- Some common idioms wrapped up
- Subclass of WWW::Mechanize (which is a subclass of LWP::UserAgent)
- Adds additional common methods
 - title_is, title_like, title_unlike
 - content_is, content_contains, content_lacks, content_like, content_unlike
- Similar to code shown before, but nicer wrapper



Fancier tests

- `page_links_ok` - Test all links on current page for 200 response
- `page_links_content_like` - follow all links and make sure all their content includes a regex
- `links_ok(\@links)` - test specific links for 200



In summary

- Perl's test harness is good and versatile
- WWW::Mechanize can be used to probe a website
- Using them together means you can test a website for health and profit
- Also look at WWW::Mechanize::Shell to automatically write Mech scripts